Facilitating Exploration of Unfamiliar Source Code by Providing 2½D Visualizations of Dynamic Call Graphs

Johannes Bohnet, Jürgen Döllner
Hasso-Plattner-Institut – University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany
{bohnet, doellner}@hpi.uni-potsdam.de

Abstract

For modifying functionality of legacy software systems developers often need to work within millions of lines of unfamiliar code. In this paper we propose a concept that exploits dynamic call graphs for (a) identifying code parts that implement the functionality to be modified and (b) guiding developers while navigating from one source code file to another. The proposed concept is implemented within a tool for analyzing complex C/C++ software systems and has been tested on various million LOC systems. The tool provides a visualization front-end that permits developers to explore the system implementation on 3 levels of abstraction: (1) source code, (2) function interaction, and (3) module interaction. A 2½D visualization view exploits perspective distortion for displaying both detailed and context information on functions and modules, by this, supporting developers during their comprehension tasks.

1. Introduction

Creation of value in business world increasingly depends on software systems that facilitate performing business processes. Developers, who need to adapt these legacy software systems to changes in IT environment or to changes in business processes, often meet the problem of being asked to do modifications within a large repository of source code that they are not familiar with – a time-consuming and cost-intensive task.

Typically requests for changes are expressed in terms of features [6]. (We use the term feature here as "system functionality that can be triggered by user interaction and produces externally visible output".) To identify feature-relevant code, a developer firstly needs to find reasonable entry points into the implementation of the feature and then start fine-grained analysis from there by reading source code. This is typically done within an IDE by switching between a set of source code files trying to understand how functions[1] interact via calls or via manipulations of data structures [13]. However, switching between files holds the risk of getting disoriented, i.e., of developers not knowing where to locate the currently seen code within a higher-level structure and of having difficulties to navigate back to already explored parts of the code. This problem arises from 2 reasons:

Firstly, it's the lack of an overview-like representation of the source code that reflects the task of trying to understand how functions interact [10]. The overview-like presentation of the module structure commonly found in IDEs (e.g., how files are organized in directories) seems to be insufficient for understanding high-level concepts of the system dynamics as it only provides a static view.

Secondly, while navigating from one source code file to another, thereby following a series of function calls, developers do not have far-reaching information on what lies beyond a specific call statement. They need to navigate first before they are able to understand what a specific call induces. This way of navigation in a *go-and-see* instead of a *see-and-go* fashion often leads to unnecessary movements that tend to disorient developers.

In this paper we present a concept for using dynamic call graphs to build up a high-level representation of source code that supports developers in locating and understanding feature implementation in unfamiliar code. The concept is implemented within a tool for analyzing complex C/C++ software systems (> million LOC). A detailed description of the tool's

[1] Throughout the paper we use the terms *function* and *module*, both coming from the world of procedural programming languages. However, in the context of this paper, the object-oriented terms *method*, *class* and *package* can be seen as synonyms for *function* and *module*; whereas a class represents the smallest form of a module.

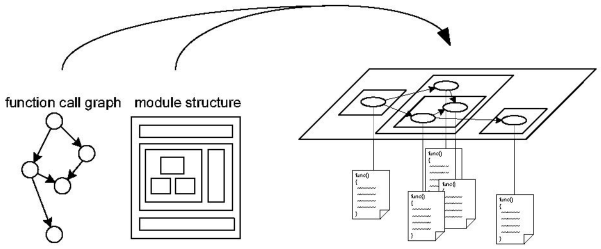


Figure 1: Function call graph and module structure are combined to build a high-level representation of feature implementation. Developers navigate through the implementation guided by the call graph, thereby accessing the functions' source code when needed.

analysis process is found in [1]. The tool provides an elaborate interactive visualization front-end that permits developers to seamlessly explore the implementation on various abstraction levels, i.e., function/module interaction and code representation.

2. Related Work

Various concepts exist to support developers in locating features in unfamiliar code. *Software Reconnaissance* [24] and the concept proposed by Eisenberg and DeVolder [7] primarily focus on *finding* code candidates. Other concepts [3, 6] also provide means of manual validation of the proposed candidates. These concepts operate on a low level of abstraction when presenting developers the analysis results – whereas the concepts of *Gadget* [8], Salah et al. [21], *Program Explorer* [14], *DJVis* [23], *E-CARES* [18], Malloy and Power [17], Jacobs and Musial [9] present their analysis results as high-level graphs showing system dynamics on various levels of abstraction. In contrast to our approach, they do not provide means of seamless navigation from high-level abstraction, such as module interaction, over function level down to plain source code, and vice versa. Additionally, except for Jacobs and Musial, no level of detail concept for optimally exploiting the display space is given.

Concepts for visualizing system dynamics [11, 14, 15, 17, 18, 19] facilitate understanding the sequence of interactions between software artifacts in time. The main idea of these 'UML sequence diagram'-like visualizations is to allocate a slice of the display space for every artifact, e.g., module or function. Within a slice, events are shown that occur during progression of time. All of the concepts have in common that, for being usefully applied, a concise set of artifacts to be shown has to be chosen beforehand.

Related work regarding the problem of orientation keeping during source code exploration is found in [4, 5, 10, 12, 20, 22]. All concepts use elaborate techniques to give developers navigation cues during exploration. However, none of them uses dynamic call graphs for this.

3. Call Graphs as Exploration Guides

Techniques for feature location have in common that they deliver suggestions which parts of the huge repository of source code are related to feature implementation and should therefore be examined in detail. This is typically done in a *go-and-see* (rather than in a *see-and-go*) fashion by analyzing the system on source code level and switching between various code files. Thereby, developers reconstruct system activity during feature execution by following function calls and conceiving data manipulations.

In our tool we combine dynamic call graphs with the module structure of the system. By this, the resulting graphs can serve as navigation guides during fine-grained exploration of the system (see Figure 1). Advantageous are:

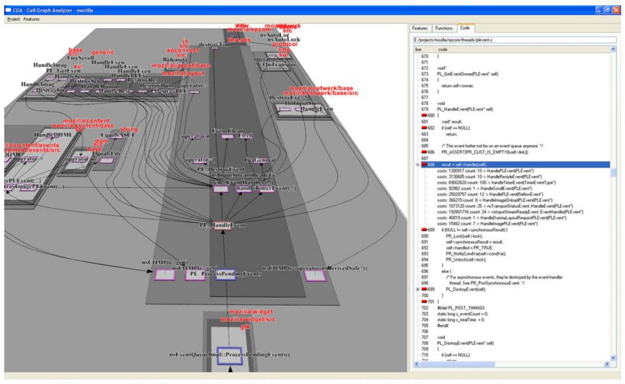
1) The source code to inspect is reduced to those parts that contribute to the feature during its execution.

2) Developers receive a high-level view on the system that enables them to better build up a mental model of how functionality is generated.

3) The flow of function calls and calls between modules gives developers hints which parts of the code should be read in detail.

4. Interactive Visualization

Our tool for locating and understanding feature implementation provides an elaborate, interactive visualization front-end that supports developers in exploring system implementation simultaneously on high-level abstraction, i.e., function and module interaction, and on low-level abstraction, i.e., code



E:/projects/mozilla/xpcom/threads/plevent.c

```
}

void*
PL_GetEventOwner(PLEvent* self)
{
    return self->owner;
}

void
PL_HandleEvent(PLEvent* self)
{
    void* result;
    if (self == NULL)
        return;

    /* This event better not be on an event queue anymore. */
    PR_ASSERT(PR_CLIST_IS_EMPTY(&self->link));

    result = self->handler(self);
costs: 1300917 count: 10 -> HandlePLEvent(PLEvent*)
costs: 3130685 count: 10 -> HandleRestyleEvent(PLEvent*)
costs: 69002620 count: 100 -> handleTimerEvent(TimerEventType*)
costs: 92982 count: 1 -> HandleScrollEvent(PLEvent*)
costs: 25029757 count: 12 -> HandlePLEvent(ReflowEvent*)
costs: 366270 count: 8 -> HandleImageOnloadPLEvent(PLEvent*)
costs: 1873120 count: 25 -> nsTransportStatusEvent::HandleEvent(PLEvent*)
costs: 192801716 count: 24 -> nsInputStreamReadyEvent::EventHandler(PLEvent*)
costs: 40819 count: 1 -> HandleDummyLayoutRequestPLEvent(PLEvent*)
costs: 15482 count: 7 -> HandleImagePLEvent(PLEvent*)
    if (NULL != self->synchronousResult) {
        PR_Lock(self->lock);
        self->synchronousResult = result;
        self->handled = PR_TRUE;
        PR_NotifyCondVar(self->condVar);
        PR_Unlock(self->lock);
    }
    else {
        /* For asynchronous events, they're destroyed by the event-handler
          thread. See PR_PostSynchronousEvent. */
        PL_DestroyEvent(self);
    }
}
#ifdef PL_POST_TIMINGS
static long s_eventCount = 0;
static long s_totalTime  = 0;
#endif

void
PL_DestroyEvent(PLEvent* self)
{
    if (self == NULL)
```

Figure 2: Graphical view and textual code view are synchronized, allowing exploring the system simultaneously on various abstraction levels.

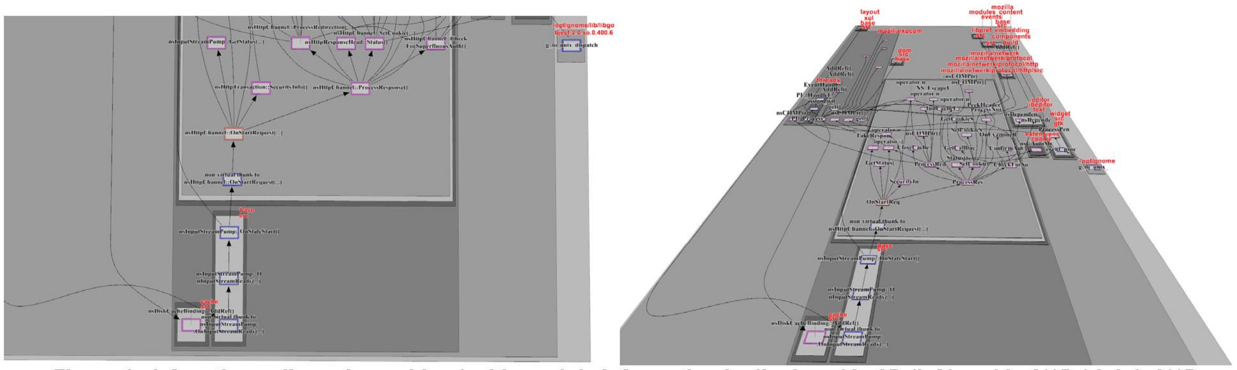


Figure 3: A function call graph combined with module information is displayed in 2D (left) and in 2½D (right). 2½D visualization allows showing more functions than in 2D, albeit most of them with low detail. However, the visualization still provides important context information, namely information on corresponding modules.

representation. Graphical view and textual code view are synchronized which allows switching seamlessly between these representations (see Figure 2).

4.1 2½D Call Graph Visualization

The graph visualization view showing function and module interaction exploits 3D perspective distortion to provide a solution for the focus & context problem, i.e., showing at the same time (a) detail information on some functions in visual focus and (b) context information, such as module information, on functions that are many calls away from the functions in focus. For this, the view displays call graphs in 2½ dimensions, i.e., on a 2D layout seen under 3D perspective (see Figure 3). This distortion technique seems to be well supported by the human processing system which in real world experience constantly decodes perspectively distorted 2D images back to the original images. Hence, although developers *see* an image of a graph whose layout constantly changes during navigation due to distortion, they *perceive* an image of a graph with a fixed layout. The constancy of the perceived 2D layout is likely to be facilitating building up a cognitive map, i.e., an internalized analogy in the human mind to the physical layout of the environment.

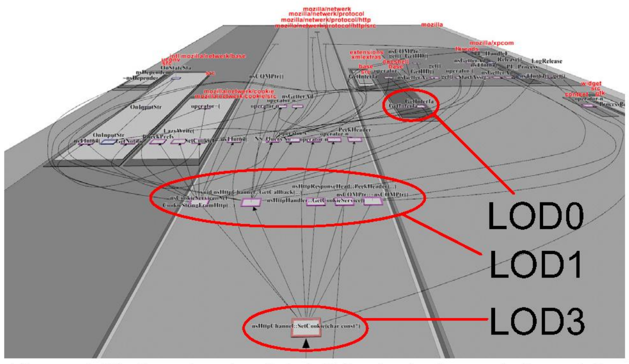


Figure 4: Various levels of detail for textual annotations. The more pixels a function shape occupies on the screen, the more details on the function name provides the annotation.

4.2 Textual Annotation with Levels of Detail

For interpretation of the graph visualization it is important to efficiently relate the visual shapes to the underlying data, namely functions and modules. For this, we attach textual annotations to the shapes. The technique's algorithm, which minimizes text occlusion, is explained in detail in [16]. Here, we shortly point out an extension of the algorithm that implements a level of detail concept for textual annotations. This concept complements the focus & context solution provided by perspective distortion. The key idea is to calculate the size that a function shape occupies in 2D pixel space. A set of threshold values then defines the level of detail for the annotation. Figure 4 illustrates how function shapes are automatically annotated with text depending on the function shapes' sizes.

5. Conclusions

When maintaining legacy software systems, developers are often confronted with the problem of performing changes within a large repository of unfamiliar source code. Extracting dynamic function call graphs from running systems seems to be of great help for developers to identify those parts of the code that implement the system functionality to be modified. Furthermore, call graphs can effectively be used to guide developers during their navigation from one source code file to another. 2½D visualization techniques seem to be well suited for exploring large call graphs. With it, perspective distortion can be exploited for implementing a focus & context concept that maps well on software engineering questions that arise during exploration of the system implementation.

Currently we are performing a user study to formally verify our findings that 2½D visualization results in better user performance than common 2D visualization [2]. As future work, we will integrate the tool within IDEs to better incorporate the tool's analysis process into existing software engineering processes.

References

[1] J. Bohnet and J. Döllner. Visual exploration of function call graphs for feature location in complex software systems. *ACM Symp. on Software Visualization*, 2006, pp. 95-104.

[2] J. Bohnet and J. Döllner. Planning an Experiment on User Performance for Exploration of Diagrams Displayed in 2½ Dimensions. *Workshop Empirische Untersuchungen von Visualisierungswerkzeugen zur Software-Analyse on Conf. on Software Engineering SE07*, 2007, pp. 223-230.

[3] K. Chen and V. Rajlich. RIPPLES: Tool for Change in Legacy Software. *Int'l Conf. on Software Maintenance,* 2001, pp. 230-239.

[4] R. DeLine, A. Khella, M. Czerwinski, and G. Robertson. Towards understanding programs through wear-based filtering. *ACM Symp. on Software Visualization*, 2005, pp. 183-192.

[5] M. Eichberg, M. Haupt, M. Mezini, and T. Schafer. Comprehensive Software Understanding with SEXTANT. *Int'l Conf. on Software Maintenance*, 2005, pp. 315-324.

[6] T. Eisenbarth, R. Koschke, and D. Simon. Locating Features in Source Code. *IEEE Trans. on Software Engineering*, 29(3), 2003, pp. 210-224.

[7] A. D. Eisenberg and K. DeVolder. Dynamic Feature Traces: Finding Features in Unfamiliar Code. *Int'l Conf. on Software Maintenance*, 2005, pp. 337-346.

[8] J. Gargiulo and S. Mancoridis, Gadget: A Tool for Extracting the Dynamic Structure of Java Programs. *Int'l Conf. on Software Engineering & Knowledge Engineering*, 2001, pp. 244-251.

[9] T. Jacobs and B. Musial. Interactive visual debugging with UML. *ACM Symp. on Software Visualization*, 2003, pp. 115-122.

[10] D. Janzen and K. DeVolder. Navigating and querying code without getting lost. *Int'l Conf. on Aspect-oriented Software Development*, 2003, pp. 178-187.

[11] D. Jerding and S. Rugaber. Using Visualization for Architectural Localization and Extraction. *Working Conf. on Reverse Engineering*, 1997, pp. 56-65.

[12] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. *Int'l Conf. on Aspect-oriented Software Development*, 2005, pp. 159-168.

[13] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Trans. on Software Engineering*, 32(12), 2006, pp. 971-987.

[14] D. B. Lange and Y. Nakamura, Object-Oriented Program Tracing and Visualization. *IEEE Computer*, 1997, 30(5), pp. 63-70.

[15] K. Lukoit, N. Wilde, S. Stowell, and T. Hennessey. TraceGraph: Immediate Visual Location of Software Features. *Int'l Conf. on Software Maintenance*, 2000, pp. 33-39.

[16] S. Maass and J. Döllner. Efficient View Management for Dynamic Annotation Placement in Virtual Landscapes. *Int'l Symp. on Smart Graphics*, 2006, pp. 1-12.

[17] B. Malloy and J. F. Power. Exploiting UML dynamic object modeling for the visualization of C++ programs. *ACM Symp. on Software Visualization*, 2005, pp. 105-114.

[18] A. Marburger and B. Westfechtel. Tools for understanding the behavior of telecommunication systems. *Int'l Conf. on Software Engineering*, 2003, pp. 430-441.

[19] L. Martin, A. Giesl, and J. Martin. Dynamic Component Program Visualization. *Working Conf. on Reverse Engineering*, 2002, pp. 289-298.

[20] C. Parnin and C. Gorg. Building Usage Contexts during Program Comprehension. *Int'l Conf. on Program Comprehension*, 2006, pp. 13-22.

[21] M. Salah, S. Mancoridis, G. Antoniol, and M. Di Penta. Scenario-Driven Dynamic Analysis for Comprehending Large Software Systems. *Int'l Conf. on Software Maintenance and Reengineering*, 2006, pp. 71-80.

[22] J. Singer, R. Elves, and M.-A. Storey. NavTracks: Supporting Navigation in Software Maintenance. *Int'l Conf. on Software Maintenance*, 2005, pp. 325-334.

[23] M. P. Smith and M. Munro. Runtime Visualisation of Object Oriented Software. *Int'l Workshop on Visualizing Software for Understanding and Analysis*, 2002, pp. 81-89.

[24] N. Wilde and M. C. Scully, Software reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 1995, 7(1) pp. 49-62.